Arquitectura y Organización

Aclaración: Toda la información fue sacada de las diapos de Alejandro Furfaro 1c 2020 y de las prácticas de mi cursada 1c 2024.

Arquitectura	**Organización**	**Hardware**
Conjunto de **recursos accesibles** para quien programa. Conjunto de: • Registros • Set de Instrucciones - ISA • Estructuras de memoria(seg y pag)	Detalles respecto a la **implementación de la ISA.** • Organización e interconexión de la memoria • Diseño de los != bloques de la CPU • Implementación de paralelismo a nivel de instrucciones y datos	Diseño lógico y tecnología de fabricación

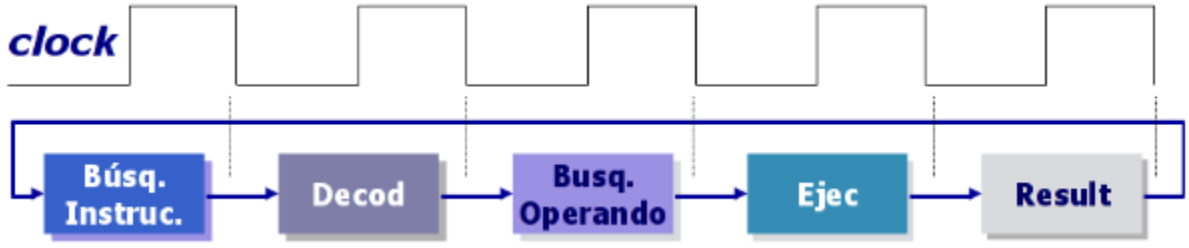
Ej. Hay procesadores con la **misma ISA y** la misma **organización** pero **distinto hardware.**

Ej. Procesadores con **misma ISA y organización diferente**. Caso AMD FX e Intel Core i7.

Microarquitectura = Implementación en el silicio(chips) de la arquitectura.
Microarquitectura = Organización + Hardware

Modelo de Von Neumann - Turing

1) Modelo de programa almacenado.
2) Datos y programas en el mismo y único banco de memoria.
3) Máquina de estados.

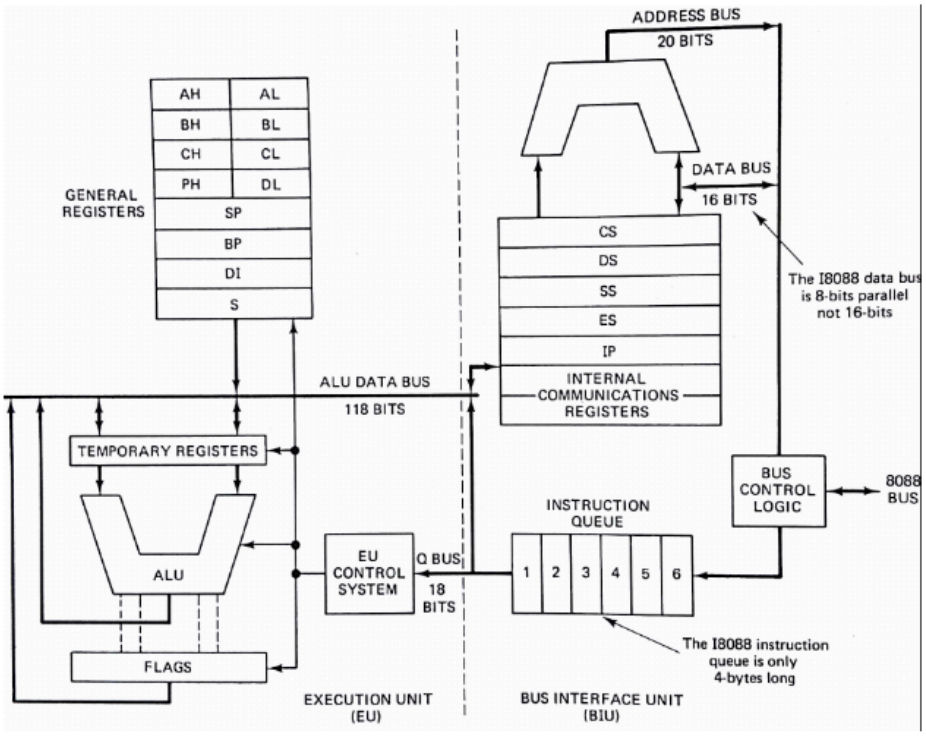


Máquina de ejecución:Cada instrucción usa varios ciclos de clock

Problema del pasado	Problema actual
Cantidad de memoria disponible	Cantidad de energía utilizada

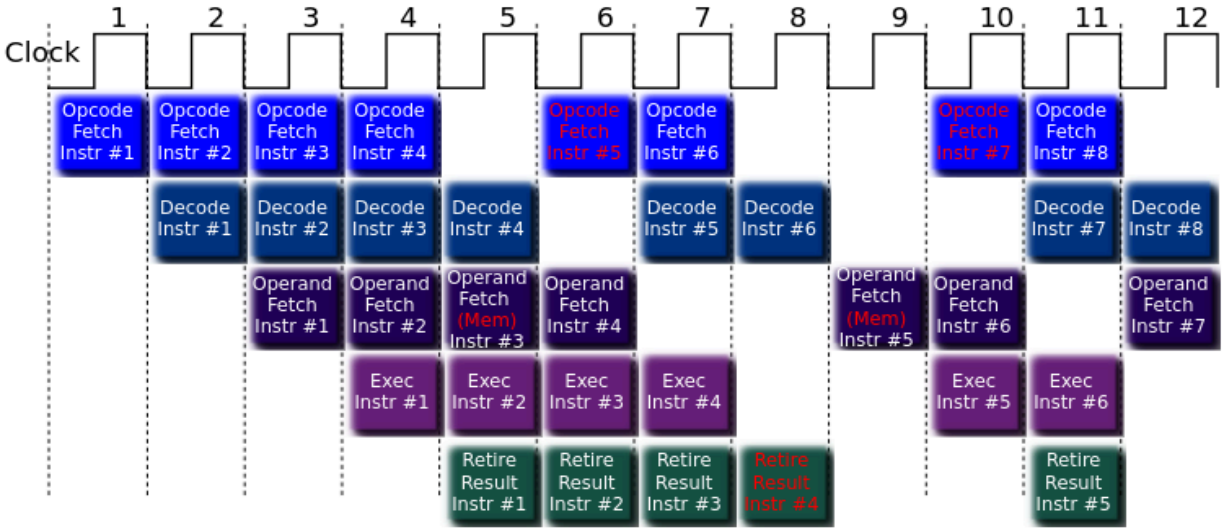
Intel cae en 1978 con su modelo x86 e IBM lo compra porque Intel le dice “yo te juro que va a tener compatibilidad hacia atrás”.

8086 Organización



Pipeline -> Instruction Level Parallelism - ILP

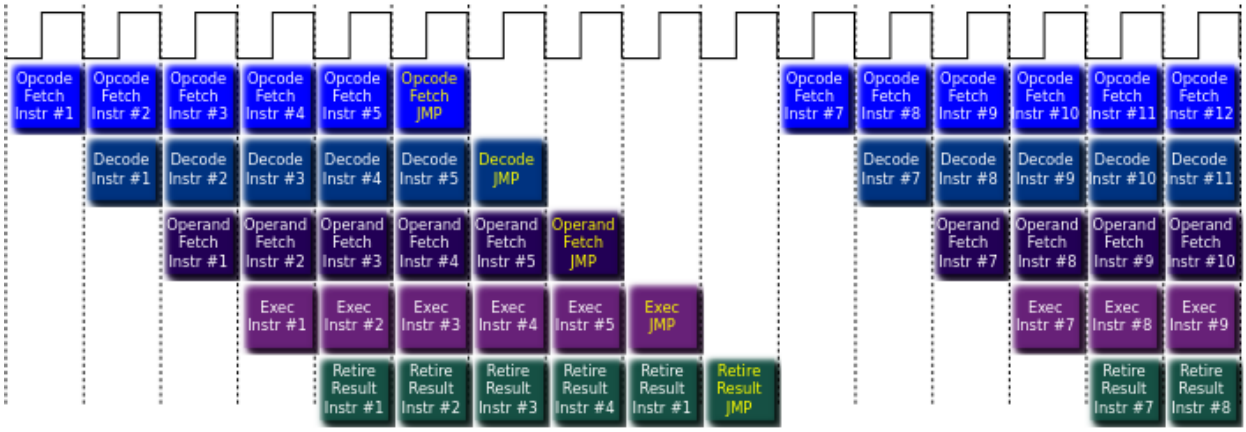
Crea el efecto de superponer en el tiempo varias instrucciones



Hay que tener cuidado porque:

- El procesador sólo tiene una etapa para acceder a memoria y compartirla para acceso a datos e instrucciones
- Si vamos a buscar un operando a memoria -> El acceso interfiere con la búsqueda de otro operando. También interfiere con el Fetch de la siguiente instrucción.

Cada obstáculo pospone una operación: CPI = CPI + 1.
Lo peor que te puede pasar es una branch, esto es una discontinuidad en el flujo de ejecución.
=> Se descarta todo lo pre procesado(branch penalty).



Hasta ese entonces todo era CISC - Complex Instruction Set Computer: Había instrucciones que hacían un montón de cosas.
En los 60, 70 nace RISC - Reduced Instruction Set Computer

Los Mandamientos RISC

1. Cantidad de registros numerosos, todos de propósito general.
2. Las instrucciones se ejecutan en un solo ciclo de clock.
3. Las instrucciones terminan en códigos de operación de igual formato y tamaño.
4. Las instrucciones deben ser sencillas de decodificar. Los números de registros deben tener la misma ubicación en los códigos de la instrucción y deben requerir la misma cantidad de bits para su decodificación.
5. No se utiliza micro código para decodificar instrucciones (no hay instrucciones complejas, como DIV o MUL). Esto evita bugs como el de **Pentium FDIV bug.**
6. Los datos en memoria se acceden mediante instrucciones simples de transferencia: LOAD y STORE.

Modos de Operación

Modo Real	Modo Protegido	Modo Mantenimiento del sistema	Modo extendido a 64 bits (IA-32e)
Modo de arranque de cualquier procesador intel IA-32 y 64 Del Modo real al modo protegido o mantenimiento 1Mb de memoria No hay privilegios Rutinas de atención para manejo de interrupciones Se pueden acceder a todas las instrucciones	Se puede implementar multitasking 4Gb de memoria 4 niveles de protección Rutinas de atención de interrupción con privilegios El acceso a instrucciones depende del nivel de protección	Se introduce este modo con los modelos 386SL y 486SL Se accede o activando una señal de interrupción #SMN o por un mensaje del APIC Se interrumpe la tarea y se pasa a un espacio separado	Intel 64 incluye IA-32 e IA-32e Modo protegido, Paginación y PAE activadas tienen que estar. IA-32e tiene dos submodos: ● Compatibilidad ● 64 bits: te habilita a usar direcciones lineales de 64 bits(siendo un sistema operativo de 64 bits)

Las instrucciones de estos procesadores pueden obtener los operandos desde:

- La instrucción en si misma
- Un registro
- Una posición de memoria
- Un port de E/S

El resultado de una instrucción se puede guardar en:

- Un registro
- Una posición de memoria
- Un port E/S

Modos de Direccionamiento

Implicito	Inmediato	Registro
En la instrucción el registro a operar está implícito y no se lo escribe. Ej MUL que actúa sobre rax	Que le pasás un inmediato en el operando Ej add rdi, 1	Operaciones donde escribís los registros en los operandos

Memoria física: Secuencia de bytes conectada al bus de address
Direcciones físicas: El espacio de direcciones que se pueden volver sobre este bus

Operando en Memoria - CISC

Intel usa **direcciones lógicas** => para **identificar a los operandos** en las instrucciones en memoria.

Dirección lógica: dirección abstracta expresada en términos de arquitectura y que después se traduce a una dirección física para el bus del sistema. Te parás en la zona que querés y después con el offset determinás a qué parte específicamente.

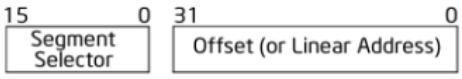


Figura: IA-32: Dirección lógica

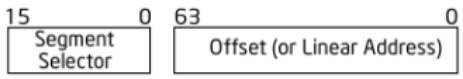


Figura: Intel® 64 : Dirección Lógica

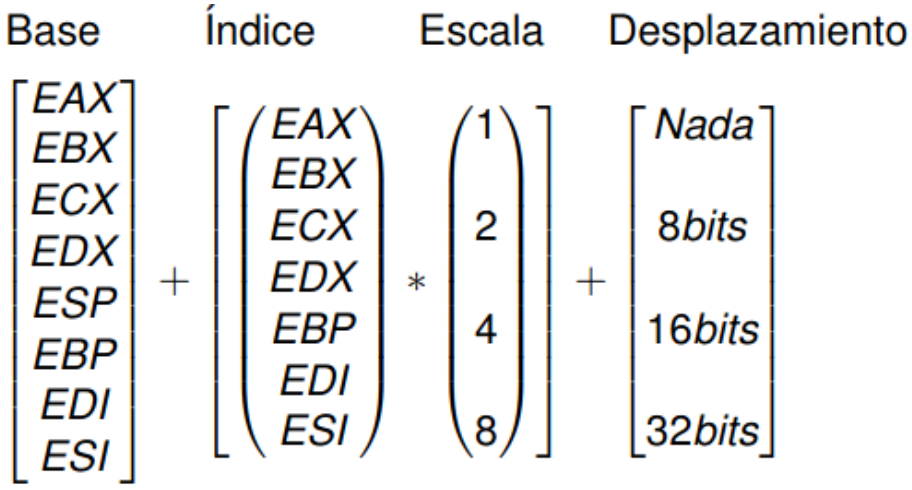
Cada dirección lógica tiene su segmento.
Si no se especifica en la dirección lógica el segmento explícitamente, el procesador lo selecciona según la tabla.

Referencia a	Registro	Segmento	Regla de selección por defecto
Instrucciones	CS	De código	Cada opcode fetch
Pila	SS	Segmento de Pila	Todos los push/pop, cualquier referencia a memoria que usa los registros ESP o EBP
Datos locales	DS	Segmento de datos	Cualquier referencia a un dato salvo que sea en el stack o destino de instrucción de string
Strings Destino	ES	De datos extra	Destino de instrucciones de manejo de strings

Desplazamiento

Desplazamiento directo	Base	índice	Escala
Puede valer 8, 16 o 32 bits y está incluído explícitamente en la instrucción	Es un valor contenido en un registro de propósito general. A partir de esta dirección se calcula el desplazamiento. Vale 32 bits en IA-32 y 64 en IA-32e	Está en un registro de prop. General, representa la dirección que queremos. Si lo incrementamos, podemos recorrer un buffer de memoria.	Valor por el que multiplicamos el valor del índice. Puede ser 2, 4 u 8.

Estas 4 cosas se pueden combinar (no necesariamente) y conseguir la dirección efectiva.
Dirección Efectiva = Base + (Indice $*$ escala) + Desplazamiento

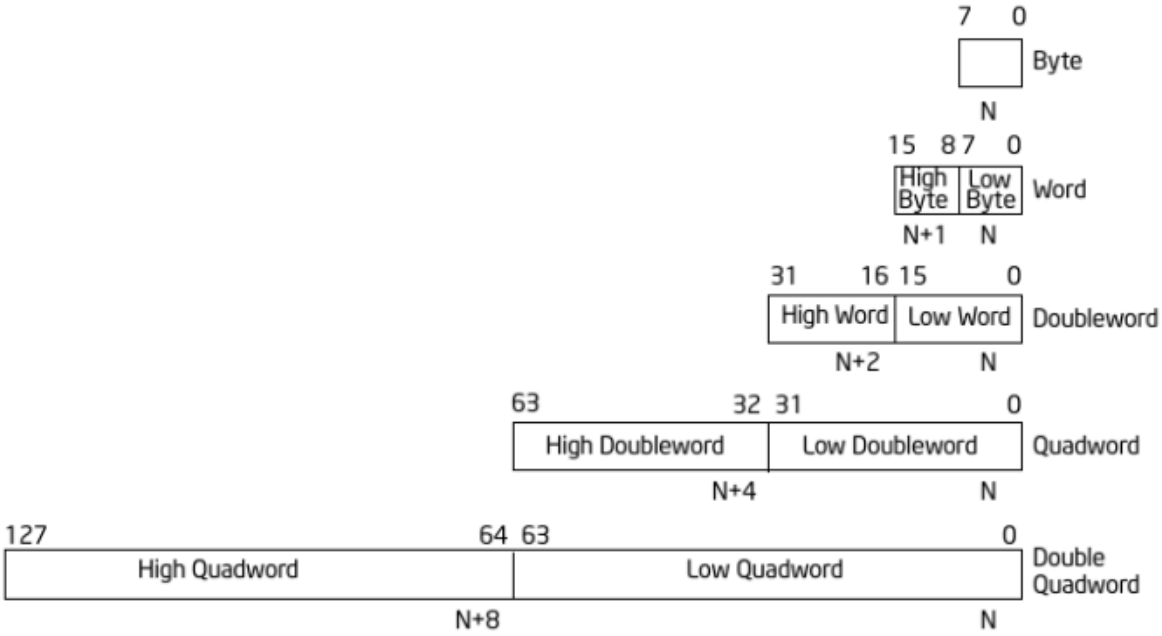


Excepciones:

- RSP/ESP NO puede usarse como índice. Su propósito es ser un puntero de pila. Otro uso es imprudente.
- RSP/ESP usan SS, el resto de los registros se asocian a DS.
- Escala se usa cuando usás un índice, sino no.

Tipos de Datos2

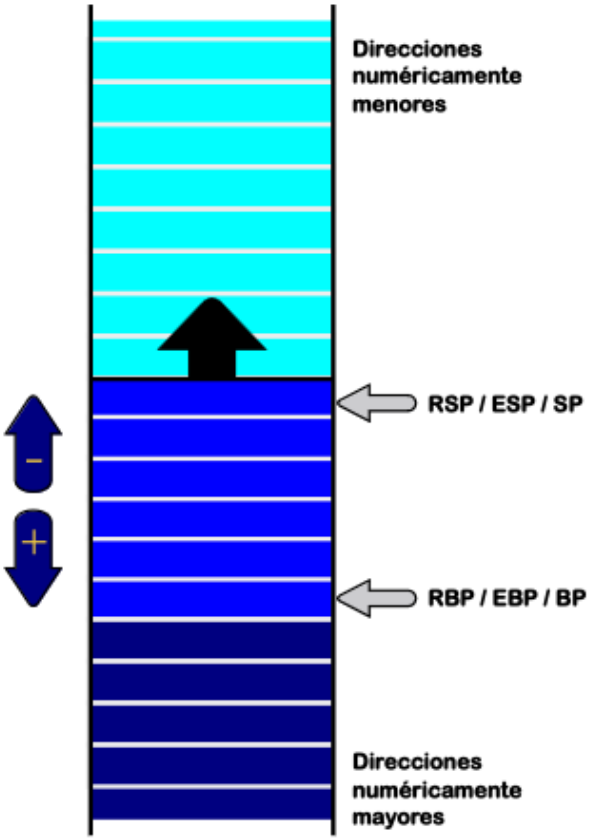
Intal usa LITTLE ENDIAN. Básicamente estás leyendo un manga.



Hay que **alinear** los datos porque -> **mejoran la performance.**
Si no están alineados, se necesitan 2 ciclos de lectura porque te quedaron los datos repartidos en en filas diferentes.

Nombres para acceder a los bits del registro en las posiciones				
63-0 (64 bits)	31-0 (32 bits)	15-0 (16 bits)	15-8 (8 bits)	7-0 (8 bits)
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

Funcionamiento básico



La **Pila** es **área de memoria contigua.**

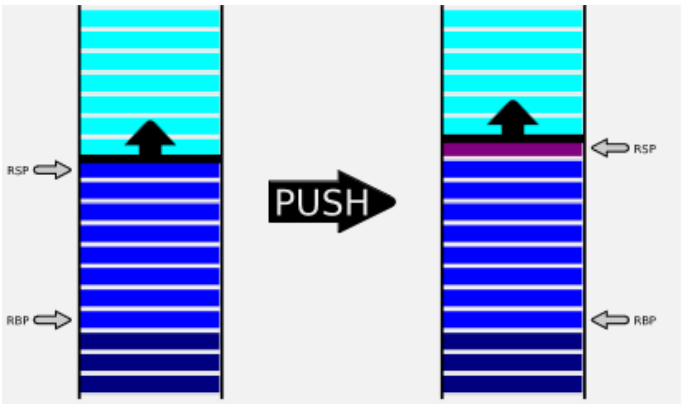
Su selector de segmento es el **SS.**

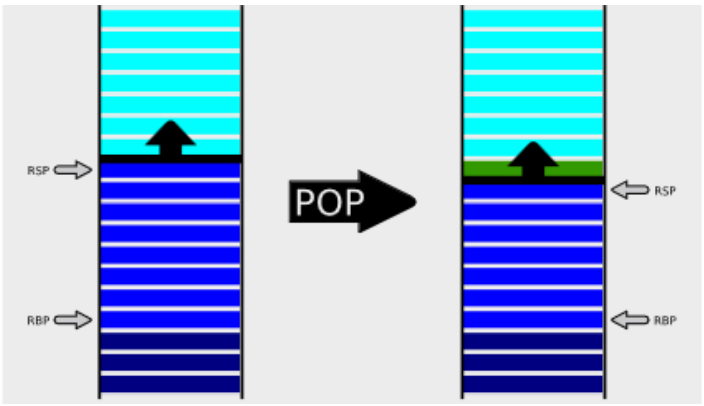
En IA-32 puede llegar hasta 4Gb de memoria cuando se usa segmentación Flat.

El segmento Pila se recorre con el SP, ESP o RSP según el modo en el que estés. 16, 32 o 64 bits.

Las imgs son de David.

Operaciones PUSH y POP. Incrementan/decrementan el RSP.
La pila crece hacia direcciones más chicas.





Uso de pila

Se usa la pila cuando:

- Usamos **CALL** en assembler.
- Cuando hay una **interrupción de hardware.**
- **INT x,** interrupción de software.
- Cuando desde **C** se invoca una función.

Alineación de la PILA

RSP TIENE QUE APUNTAR A DIRECCIONES DE MEMORIA ALINEADAS DE ACUERDO A SU ANCHO DE BITS.

Ej. ESP alineado a double words(32 bits)

- El tamaño de cada elemento en la pila es igual al modo de trabajo del procesador. Ej. **PUSH AL** consume 16, 32 o 64 bits dependiendo del segmento. **Nunca consume 8 bits.**
- El **incremento/decremento** del stack pointer se corresponde con el tamaño del segmento. (2, 4 u 8 BYTES).

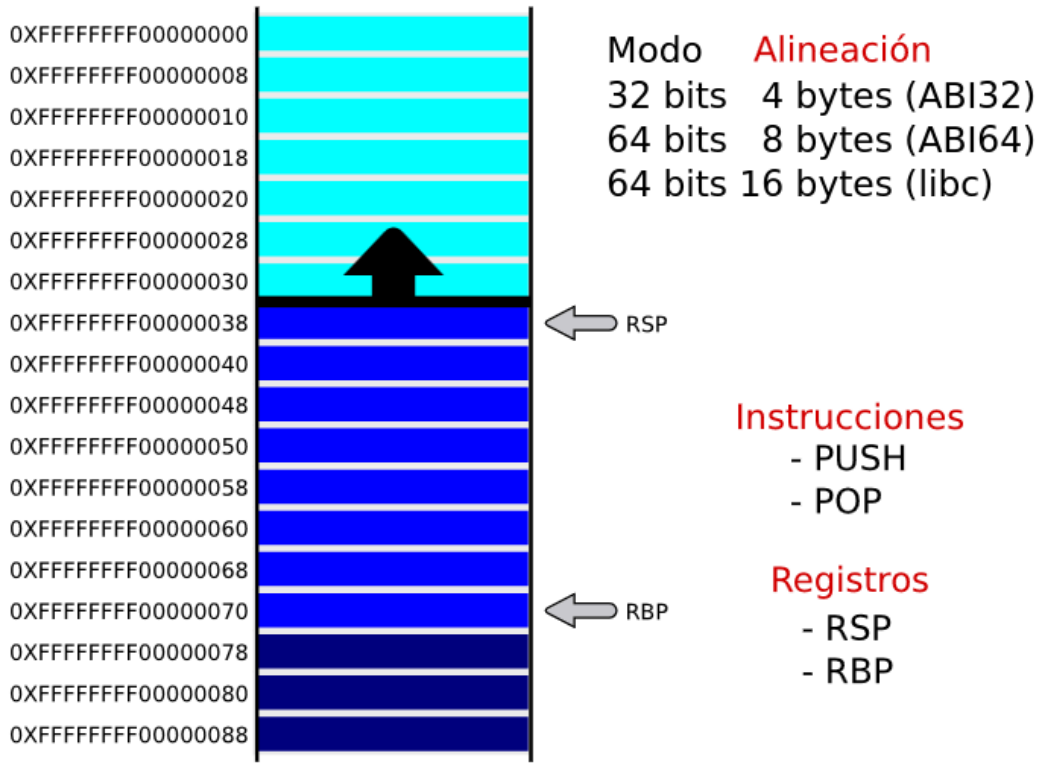


Figura: Alineación según el modo de trabajo.©David

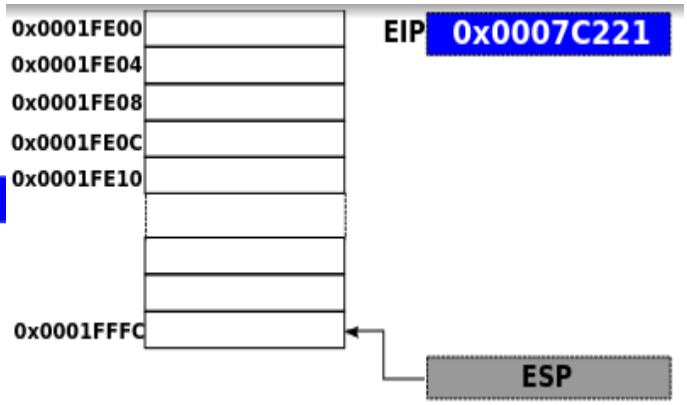
Llamado Near - mismo segmento

Los jugadores: EIP y ESP

Dicen que es similar a un debugger pero digamos que tampoco entiendo mucho los debugger (? así me va. Sigamos.

Tenemos el siguiente código con esta pila

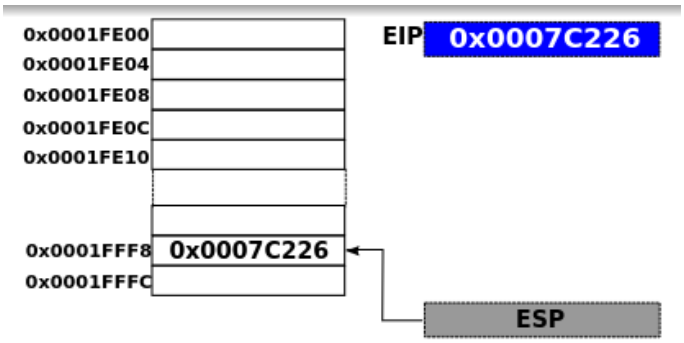
```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
call setmask ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask       ;aplica la mascara
   ret               ;retorna
```



ESP apunta a la base de la pila. EIP a la instrucción a ejecutar.

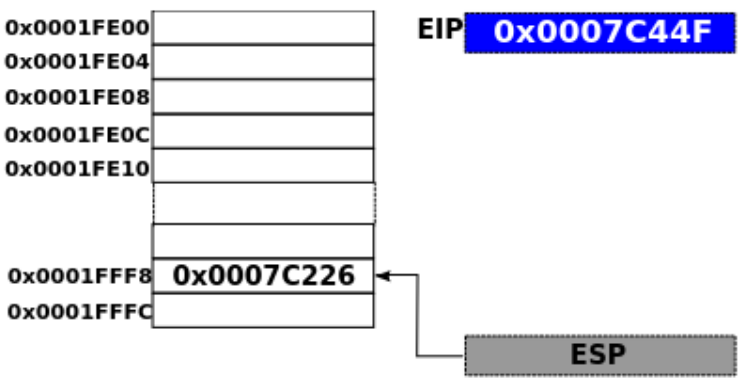
El CALL es 1 byte de operación y 4 bytes de dirección efectiva(estamos en 32 bits)
Una vez que ejecutamos el “in ax, dx”, el EIP se incrementa 5 bytes que es donde está la siguiente instrucción después del CALL.

El EIP queda 0x0007C226



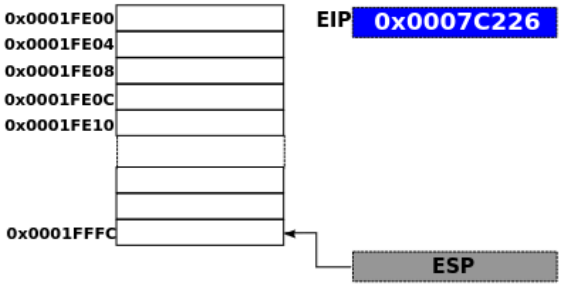
El procesador GUARDA este EIP en el stack.

Para saber a dónde saltar, el procesador lee la dirección efectiva del setmask y lo **carga** en el EIP.



Para volver de la rutina setmask
Usamos el ret
Que recupera de la pila el eip

- Finalizada la ejecución de **ret** estamos otra vez en el código llamador.
- Pero en la instrucción siguiente a CALL

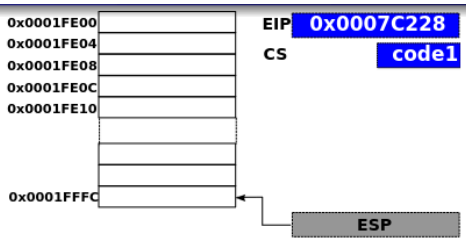



```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx      ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
...
   ...
setmask:
   and ax,mask     ;aplica la mascara
   ret             ;retorna
```

Llamado Far - El destino está en otro segmento

Jugadores: ESP, EIP, CS
Hay que memorizar: EIP y CS para saber a dónde volver. Tanto en instrucción como en segmento.

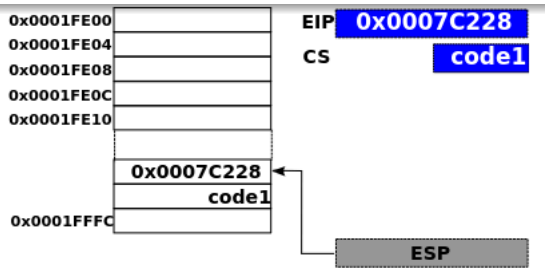
- Ahora la instrucción mide 7 bytes ya que se agrega el segmento
- Por lo tanto el EIP se incrementa 7 lugares
- Y se memoriza en la pila la dirección FAR.



NOTA:
Siempre antes de almacenar en la pila, debe antes decrementar el valor del ESP.

```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx      ;lee port
   call  code2:setmask  ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and    ax,mask          ;aplica la m scara
   retf                    ;retorna
```

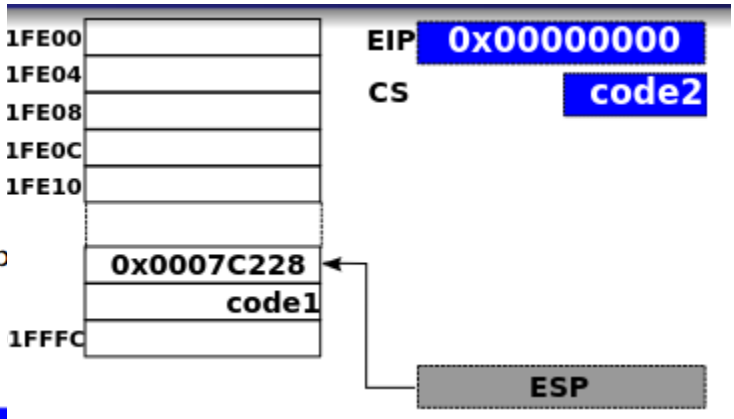
- Luego del valor del segmento guarda en la pila, el valor de EIP al cual debe retornar, y que lo llevará a buscar la siguiente instrucción al CALL.
- Decrementará nuevamente el valor del ESP, antes de almacenar.




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx      ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascar
   ...
section code2
setmask:
   and    ax,mask          ;aplica la m scara
   retf                    ;retorna
```

La dirección de la rutina *setmask* es code2:offset
Como está ni bien al comienzo del segmento, su offset es 0x00000000

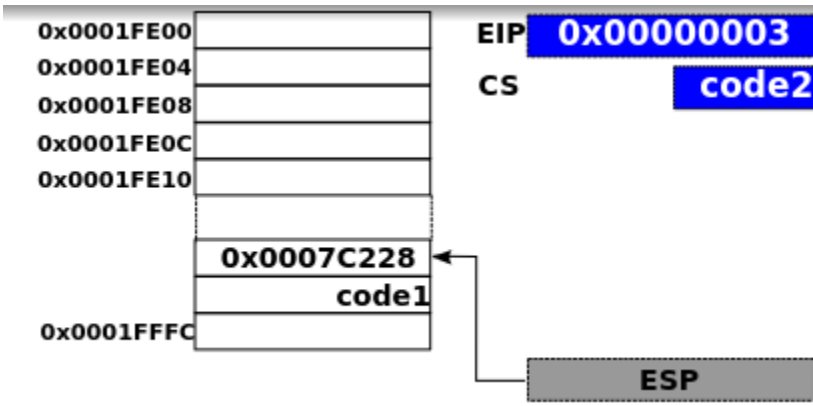
```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx      ;lee port
   call  code2:setmask ;llama a sub
   ...
section code2
setmask:
and ax,mask ;aplica la máscara
   retf                    ;retorna
```



<= La pila correspondiente a este código

Para retornar de un call far hay que recuperar el segmento y el offset. [link a una pregunta sobre cómo sabe el procesador cuando es un call far o near]

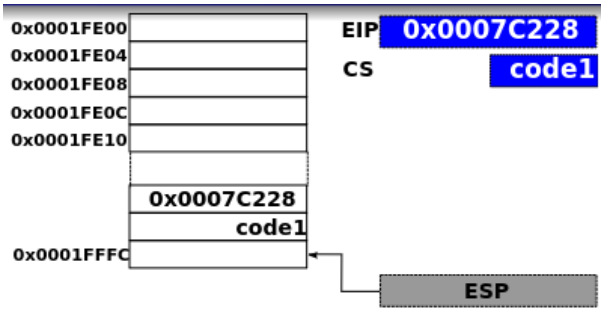
```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx      ;lee port
   call  code2:setmask ;llama a
   ...
section code2
setmask:
   and    ax,mask           ;aplica l
retf ;retorna
```



Se retorna con retf (PARA ESA EPOCA)
Se recupera la dirección efectiva, decrece el SP

Se recupera el CS, decrementa y volvemos a la siguiente instrucción después de haber hecho el call far

```
...
mov   dx,0x300
in    ax,dx      ;lee port
call  code2:setmask ;llama
...
section code2
setmask:
```

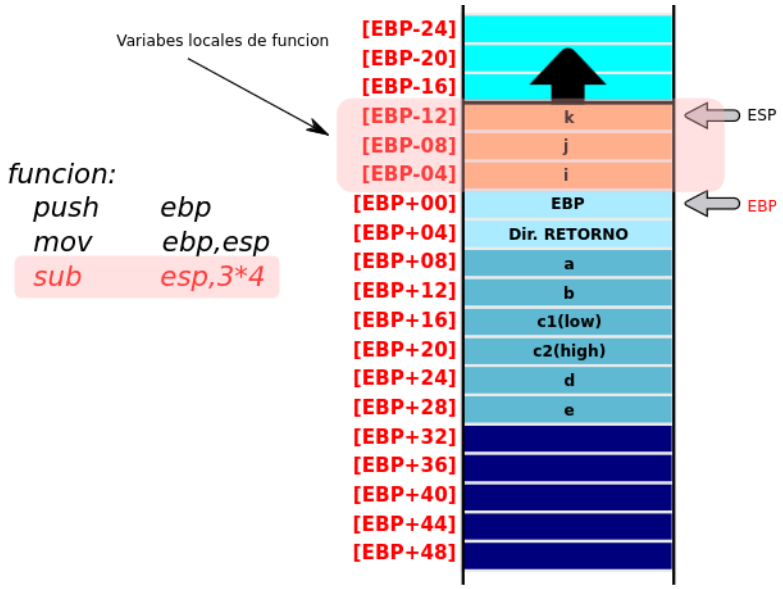


Interrupciones

Acá guardamos la dirección de retorno y el estado del procesador: las eflags.
La dirección de retorno es far y en general es multitasking(cada tarea tiene su pila nivel de kernel) -> guardamos el cs
La instrucción de retorno es iret

Convención llamadas en C - Modo 32 bits

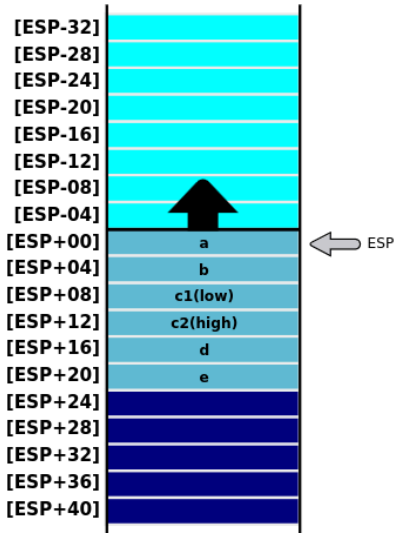
int f1(int a, float b, double c, int* d, double* e)



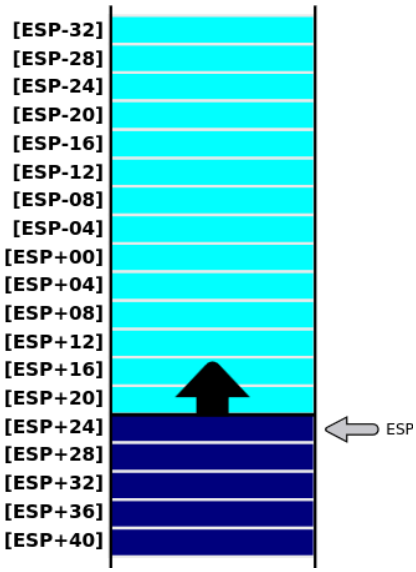

```
funcion:
   push     ebp
   mov      ebp,esp
   sub      esp,3*4
```

C1 y c2 forman el parametro C. Se pushean de derecha a izquierda.

```
Llamando:
   .............
   push     e
   push     d
   push     c2
   push     c1
   push     b
   push     a
   call     funcion
   add      esp,6*4
```




```
Llamando:
   .............
   push     e
   push     d
   push     c2
   push     c1
   push     b
   push     a
   call     funcion
   add      esp,6*4
```



Interacción C-ASM

ABI

- Los registros se usan en orden dependiendo del tipo
- Los registros de enteros guardan parámetros de tipo Entero o Puntero
- Los registros XMM guardan parámetros de tipo Flotante
- Si no hay más registros disponibles se usa la PILA
- Los parámetros en la PILA deben quedar ordenados desde la dirección más baja a la más alta.

Los resultados van en RAX, si son floats o doubles en **xmm0/1**

Tareas

Estructuras en juego:

- **TSS de cada tarea** - es un espacio en memoria donde voy a guardar el contexto de ejecución y pilas de privilegio, tamb CR3
- **Descriptores de TSS** en la GDT
- **TR** - task register, contiene el selector de segmento para ubicar al descriptor de la TSS en la GDT de la tarea que se está ejecutando actualmente

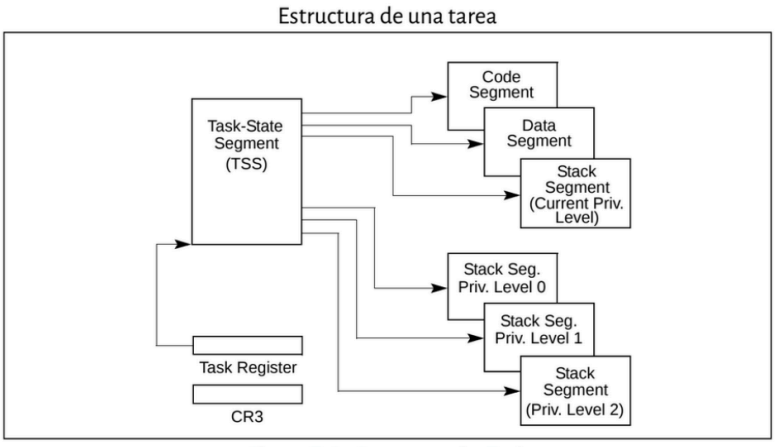
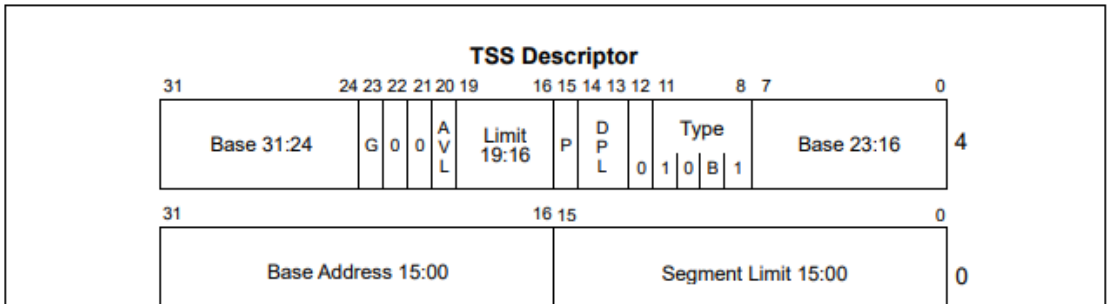
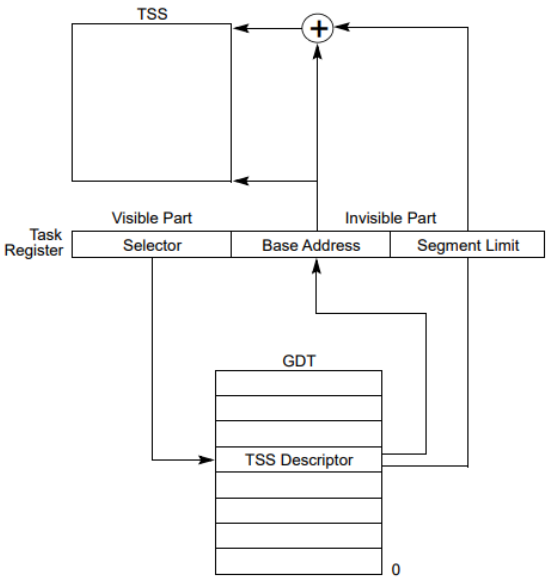


Figure 7-1. Structure of a Task



Scheduler de tareas

-cuando todas las tareas se ejecutaron al menos una vez-En el parcial podemos decir "Ya todas las tareas se ejecutaron al menos una vez"



- Para saltar a una tarea cualquiera es necesario poder **modificar** el selector de segmento.
- Usando: `jmp <selector>:0`. No es posible, ya que `<selector>` es fijo.
- Entonces, se debe usar memoria para indicar el selector de segmento para el intercambio.

Se define en algún lugar del código la siguiente estructura:

```
offset:   dd 0
selector: dw 0
```

La estructura definida se puede ver como una dirección lógica de 48 bits en *little endian*

En la rutina se utiliza de la siguiente forma:

```
...
mov [selector], ax ; se carga el selector de segmento
jmp far [offset]   ; se salta a la direccion logica definida
...
```

```
_isr32:
   pushad
   call pic_finish1
   call sched_nextTask
   str cx
   cmp ax, cx
   je .fin
      mov [selector], ax
      jmp far [offset]
   .fin:
   popad
 iret
```

str cx carga el valor del TR en cx

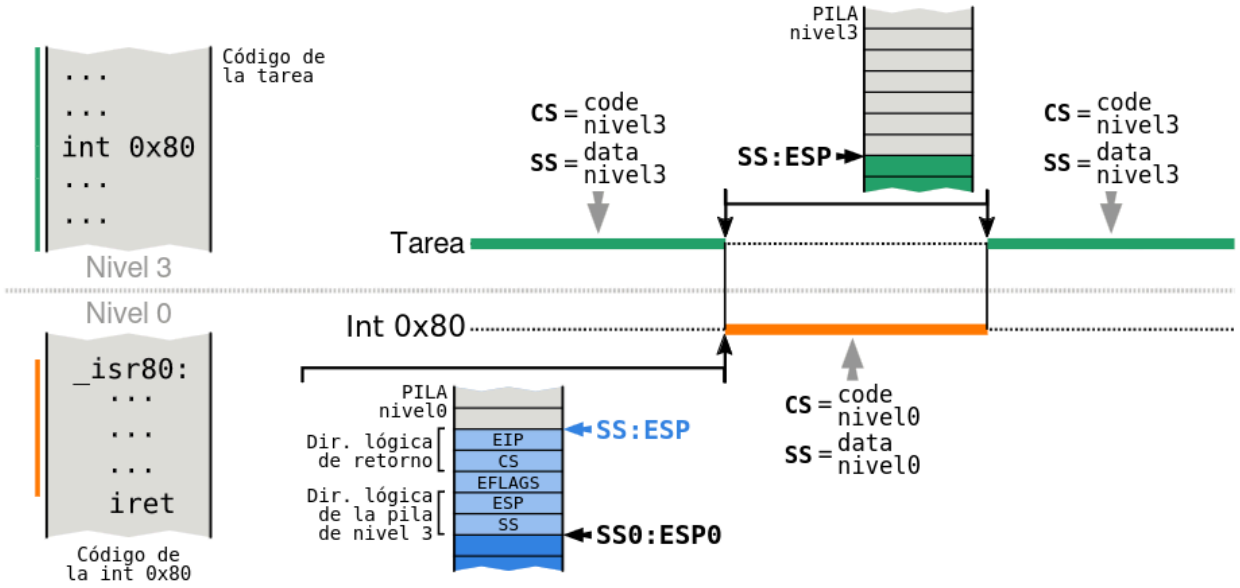
Tarea Inicial

Ni bien arranca el procesador, hay que cargar a mano la tarea inicial. Esto es Cargar el TR a mano para que el procesador se entere que esa es la tarea que está ejecutando.

Tarea Iddle

Cuando nos quedamos sin tareas para ejecutar, se usa esta tarea que no hace nada.

Cambios de privilegios al cambiar una tarea - interrupciones



Desde nivel cero NO podemos usar la pila de nivel 3 para guardar el estado de retorno y variables locales. Por lo tanto se debe cambiar la base de pila. La nueva base de la pila se toma desde los campos SS0:ESP0 en la TSS.

El estado de la pila de nivel 3 se guarda en la pila de nivel 0. **Antes de hacer pushad o después del popad estamos en nivel 3.**

Paginación

Paginacion tambien se activa. CR0.PG = 1

Interrupciones

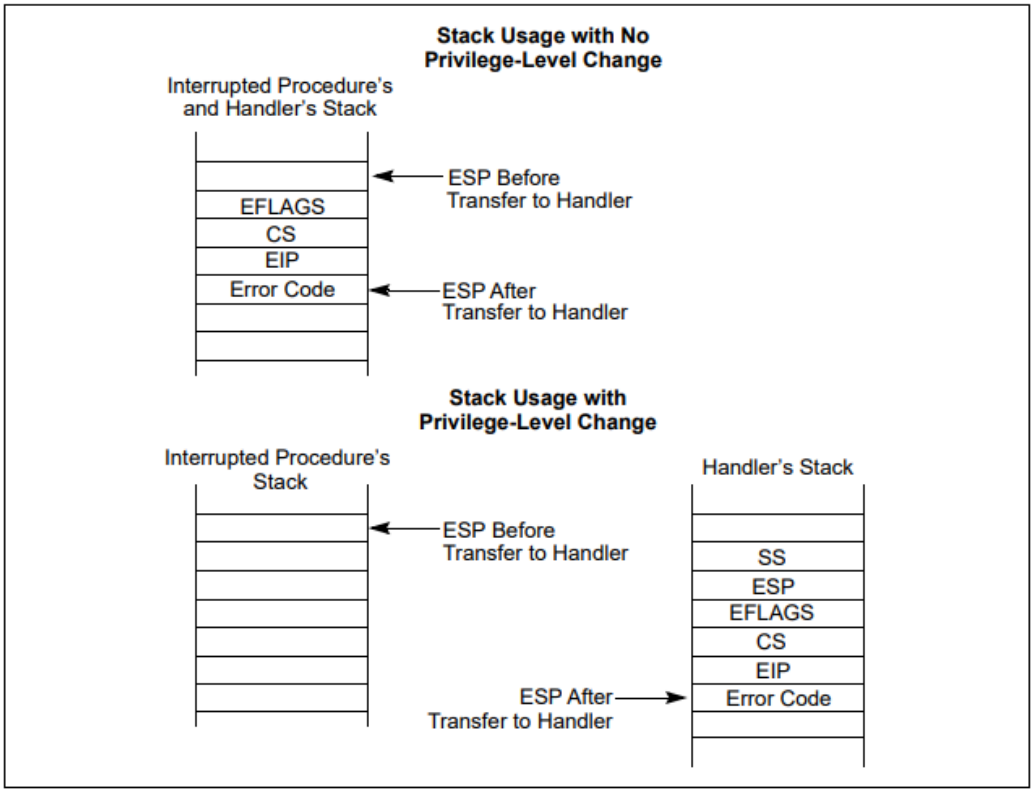


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines El error code es opcional

Segmentación

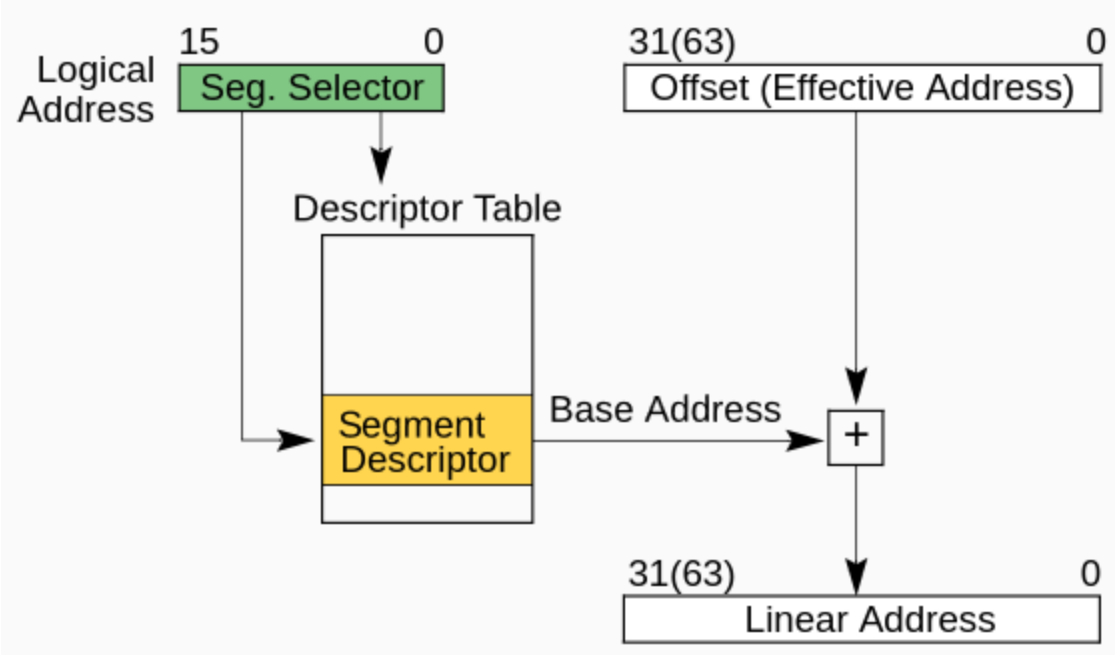
Modo Protegido **cargar la GDT**

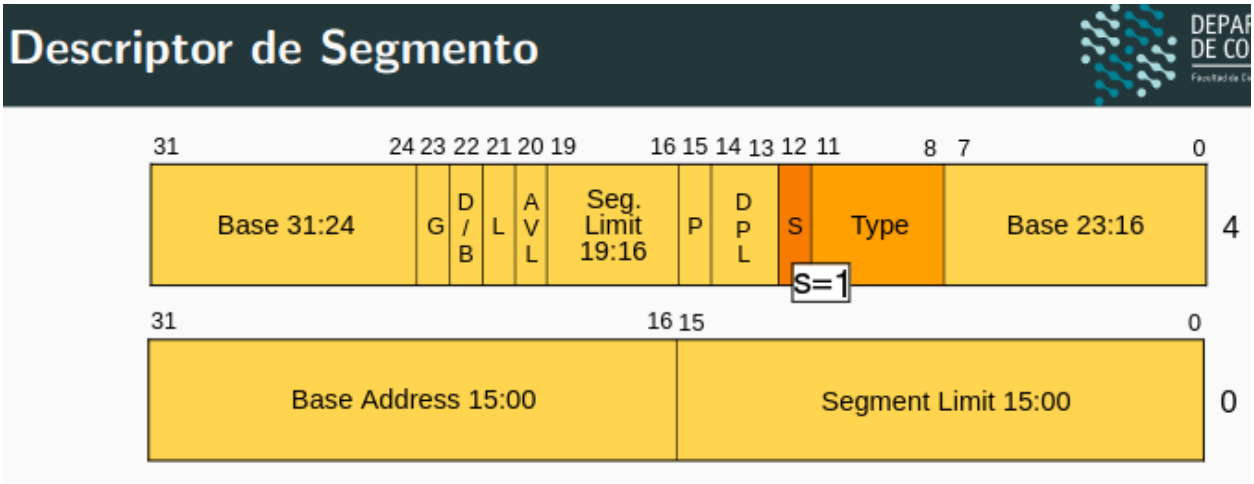
```
lgdt [GDT_DESC] ;cargamos la GDT

mov eax, cr0
or eax, 1 ; pongo en 1 el bit menos significativo de CR0, q es PE
mov cr0, eax ; lo muevo de nuevo a cr0
jmp CS_RING_0_SEL:modo_protegido
   ;CS_RING_0_SEL es un descriptor de segmento de tipo codigo nivel 0
```

Para pasar a modo protegido

- ☐ CR0.PE = 1





Notas generales para mí

- Siempre chequear el bit de P en caso de ser necesario
- La pila crece para abajo [esp + 1] y ese 1 son 4 byte
- La pila tiene que estar alineada a 4 bytes
- Cada PDE maneja hasta 4Mb - 400 000B (si estás en una direccion más grande que eso, necesitás otra entrada en la PDE)
- Cada PTE maneja una página de 4kb
- Si quiero definir variables temporales
- .data
  Variable dd 0 -> me crea una variable double 32 bits en memoria llena de ceros

Y yo en el codigo la puedo usar como [variable]

- Si pusheas cosas a la pila despues del pushad, asegura de hacer el add esp, x bytes necesarios para moverte hasta el eip necesario
- Si el eip tenia un error, movete [eip + bytes] necesarios hasta la siguiente instruccion (ej parcial de maca)